CSCI 3155: Principles of Programming Languages
Exam preparation #1
2007

Exercise 1. Consider the "if-then-else" construct of Pascal, as in the following example:

```
IF   1 = 2
THEN PRINT "X"
ELSE PRINT "Y"
```

(a) Assume that the "if-then-else" is statement, i.e., *one* possible production of the nonterminal $\langle stmt \rangle$ (you don't have to specify any others), and both branches of the conditional should again allow statements. The condition is an arbitrary expression, expressed as the nonterminal $\langle expr \rangle$.

Write a BNF grammar for the above.

(b) Add another rule that allows programmers to omit the `ELSE` branch. Do not use an ε construct.

(c) Is your grammar ambiguous? If so, alter it so that it is no longer ambiguous. If not, explain why it is not ambiguous.

Exercise 2. Consider the following BNF grammar of an expression syntax, with the start symbol $\langle A \rangle$:

$$
\begin{array}{lcl}
\langle A \rangle & \rightarrow & \langle B \rangle \ + \langle C \rangle \\
& | & \mathit{var} \\
\langle B \rangle & \rightarrow & \langle D \rangle \ * \langle E \rangle \\
& | & \langle E \rangle \\
\langle C \rangle & \rightarrow & \langle A \rangle \\
\langle D \rangle & \rightarrow & \langle B \rangle \\
\langle E \rangle & \rightarrow & (\ \langle A \rangle\ ) \\
& | & \mathit{var}
\end{array}
$$

(a) Is the grammar ambiguous? If so, show that it is. If not, explain.

(b) Determine the associativity of "+" and "*", and explain.

Exercise 3. In the programming language LISP, all definitions and expressions are constructed from so-called S-Expressions. An S-Expression can be one of three things: A simple name, a "pair", or a function call.

A pair is a pair of parentheses containing an S-Expression followed by a period (".") and another S-Expression.

A function call is a pair of parentheses containing a sequence of S-Expressions; this sequence must not be empty.

Below are examples of LISP expressions:

```
(defun tuple (a b) (a . b))

(defun mapcar (f l)
   (if (null l)
       nil
       (cons (funcall f (car l)) (mapcar f (cdr l)))))
```

A LISP program is a list of S-Expressions. A LISP program must not be empty.

(a) Assume a terminal symbol *name* that describes the class of LISP names. Describe the syntax of LISP programs.

(b) Addition in LISP can take arbitrarily many arguments, as in

$$(+\ 1\ 2\ (+\ 3\ 4\ 5)\ 6)$$

Give the denotational semantics of LISP addition.

(c) All LISP programs are constructed as above. There are no formal control structures, only special names that happen to have the semantics of control structures. Give a critique of this syntax, keeping in mind that LISP is nowadays seen as a general-purpose programming language.

Exercise 4. Consider the following program:

```
VAR y : INTEGER;
PROCEDURE P (x : INTEGER) : INTEGER =
BEGIN
    y := y + x;
    IF 15 > y
    THEN VAR x : [0 TO 15]
        BEGIN
            x := y;
            y := P(y + 1);
            RETURN x + y;
        END
    ELSE RETURN 2;
    END
END
```

Assume static scoping.

(a) Determine the scopes of the paramter `x` and the global variable `y`.

(b) Determine the lifetimes of variables `x` and `y`.

Exercise 5. For the program from the previous exercise, list four different bindings that affect variable "`x`".

Exercise 6. (**Tricky!**) Assume the following program:

```
VAR x : INTEGER;
VAR r : INTEGER;

PROCEDURE P(y : INTEGER; z : INTEGER) =
VAR v : INTEGER
BEGIN
    z := r;
    v := y + y;
    x := x + v + z;
END;

PROCEDURE Q(y : INTEGER) : INTEGER =
BEGIN
    x := x + 1;
    RETURN y;
END;

PROCEDURE F(x : INTEGER; y : INTEGER) : INTEGER =
BEGIN
    IF y > x
    THEN RETURN x
    ELSE BEGIN
             x := x + F(x, y + 1);
             RETURN x
         END
    END
END;

BEGIN
    x := 1;
    r := F(2, 0);
    VAR r : INTEGER
    BEGIN
        r := 0;
        P(Q(0), x);
    END;
    PRINT x
END
```

Assume stack-dynamic storage. Choose a parameter passing mode and scoping to make the program print a certain result, and explain:

(a) 2
(b) 4
(c) 8
(d) 16
(e) 32

Exercise 7. A language designer proudly shows you the subtyping rules he came up with for his language:

$$\begin{array}{rcl} \mathsf{word} & <: & \mathsf{short} \\ \mathsf{int} & <: & \mathsf{long} \\ \mathsf{double} & <: & \mathsf{long} \\ \mathsf{int} & :> & \mathsf{short} \\ \mathsf{word} & :> & \mathsf{long} \\ \mathsf{long} & :> & \mathsf{double} \end{array}$$

"Now, some of these may seem a bit debatable, but trust me, I have my reasons to choose these." The language designer then explains that he wants to allow implicit widening conversions throughout the program.

(a) What can you tell about the type system, from the above rules?

(b) Can you find a simpler description of the type system? If so, give it below. If not, explain why the above cannot be simplified.

Exercise 8. Some programming languages, such as C++, allow both pointers and references. In languages that have explicit pointers or references, we can think of both as type constructors; in terms of C++, "`*`" is a pointer type constructor, and "`&`" is a reference type constructor.

Due to orthogonality, we can apply both type constructors to results from each other.

Explain the difference between a *pointer to a reference* and a *reference to pointer*.

Exercise 9. Following up on our discussion of subtyping in class, we can also allow subtyping for arrays.

Consider the following MYSTERY program fragment:

```
TYPE ATYPE = ARRAY AR OF AT;
TYPE PTYPE = ARRAY PR OF PT;

VAR A : ATYPE;
...
PROCEDURE F(P : PTYPE) =
...
```

(a) Assume Pass-by-value. When should `ATYPE` and `PTYPE` be subtypes of each other? Explain why your choice is natural, i.e., at most requires information to be discarded.

(b) Assume Pass-by-value-result. When should `ATYPE` and `PTYPE` be subtypes of each other? Explain why your choice is natural, i.e., at most requires information to be discarded.

Exercise 10. Python is a language with implicit heap-dynamic variables. As we discussed in class, Python uses indentation to indicate block structure. Consider the following Python program fragment:

```
v = read_some_input()
if v > 0:
        k = compute_something(v)
        if k < 0:
                k = "Result is undefined."
else:
        k = "Input is negative."

if isinstance(k, str):
        print "Error: " + k
else:
        print ("Result: ", k)
```

Note the use of `isinstance(v, str)` to determine whether the type of `k` is a string (indicating an error) or not.

(a) Describe, abstractly, how you could re-write this code to use unions.

(b) Would tagged or untagged unions be more appropriate? Explain.

Exercise 11. A language designer decides to introduce *combined subrange types*, i.e., types that describe values out of a list of subranges, to the language she is working on. Such combined subrange types allow us to describe types such as `([1 TO 4],[7 TO 8], [100 TO 102])`: A value v is of this type iff any one of the following holds:

$$\begin{aligned} 1 \leq &\ v &\leq 4 \\ 7 \leq &\ v &\leq 8 \\ 100 \leq &\ v &\leq 102 \end{aligned}$$

Consider the following combined subrange types:

(a) Which of the following types should naturally be subrange types of each other? Annotate appropriately.

`([1 TO 7])`	`([2 TO 5])`
`([17 TO 22],[39 TO 141])`	`([94 TO 97],[99 TO 100],[107 TO 111])`
`([1 TO 713],[715 TO 982])`	`([512 TO 982])`
`([52 TO 54])`	`([61 TO 115],[31 TO 55])`

(b) All of the above types were valid examples of combined subrange types. Discuss the utility of these combined subrange types, given our criteria.

Exercise 12. A friend shows you the following program and mentions that it doesn't typecheck; it died on the line marked `(* error *)` during dynamic type checking. Your friend knows that the language uses pass-by-reference for arrays, but is otherwise somewhat unfamiliar with its semantics.

```
TYPE T = ARRAY [0 TO 99] OF INTEGER;
VAR B : ARRAY [0 TO 99] OF INTEGER;
VAR A : T;
VAR I : [0 TO 99];

PROCEDURE P(X : ARRAY [0 TO 99] OF INTEGER) =
BEGIN
    IF 100 > X[1] AND 100 > X[X[X[1] + 1]]
    THEN X[X[1]] := X[X[1]] + X[X[X[1] + 1]];
    ELSE RETURN;
    END
END

BEGIN
    I := 0;
    WHILE (100 > I) DO
        A[I] := I;
        I := I + 1;
    END;
    B := A;
    P(A);       (* error *)
    I := 0;
    WHILE (100 > I) DO
        PRINT A[I] + B[I];
    END
END
```

Give the most plausible explanation for the type error.

Exercise 13. What is the difference between static type checking and static type binding? Explain.

Exercise 14. Consider the following BNF grammar with start symbol $\langle S \rangle$:

$$
\begin{array}{lcl}
\langle S \rangle & \rightarrow & \langle L \rangle \mid \ \mathbf{!}\ \langle A \rangle\ \langle L \rangle\ \mathbf{!} \\
\langle L \rangle & \rightarrow & \langle A \rangle \mid \langle A \rangle\ \langle L \rangle \\
\langle A \rangle & \rightarrow & \langle A \rangle + \langle B \rangle \mid \langle B \rangle \\
\langle B \rangle & \rightarrow & \mathbf{a} \mid \mathbf{!}\ \langle L \rangle\ \mathbf{!} \mid \mathbf{b}
\end{array}
$$

(**Skills 2.2, 2.5**; tricky!) Is this grammar ambiguous? If so, prove that it is ambiguous. If not, explain why you believe that it is not, by analogy with the examples used in the book for discussing ambiguity.